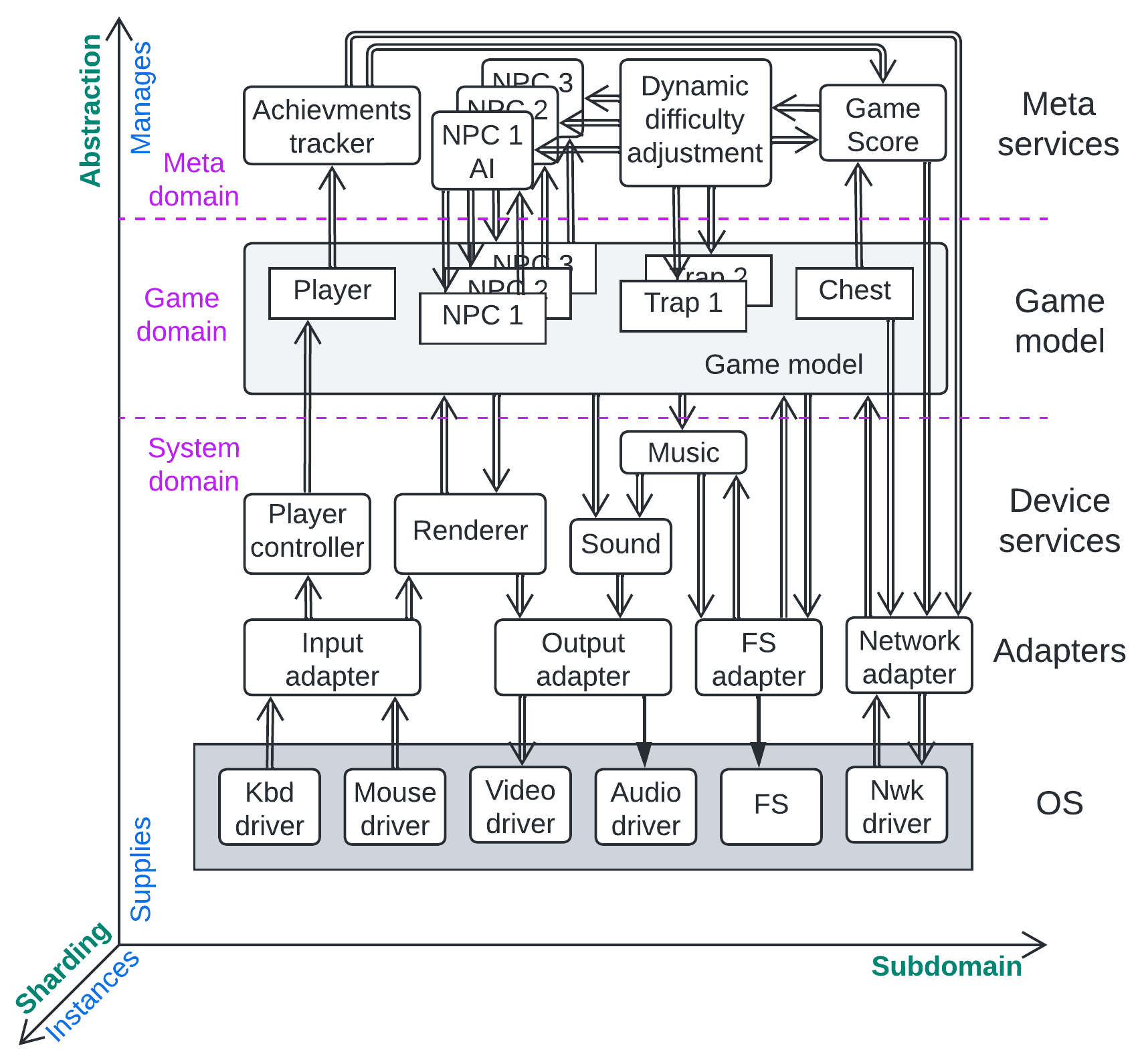
*Part* 1 2 **3** 4 5

Introduction to Software Architecture with Actors:  
Part 3 – On Simple Systems

After having looked into the ways events are processed inside individual actors, it is time to try combining several actors (or splitting an actor by messaging interfaces).

## The Coordinate System



The remaining parts of the publication examine the relation between structural diagrams (drawings of components and their interactions) for common types of systems of actors/services and the properties of those systems. The following coordinates, abbreviated as ***ASS*,** will be used consistently:

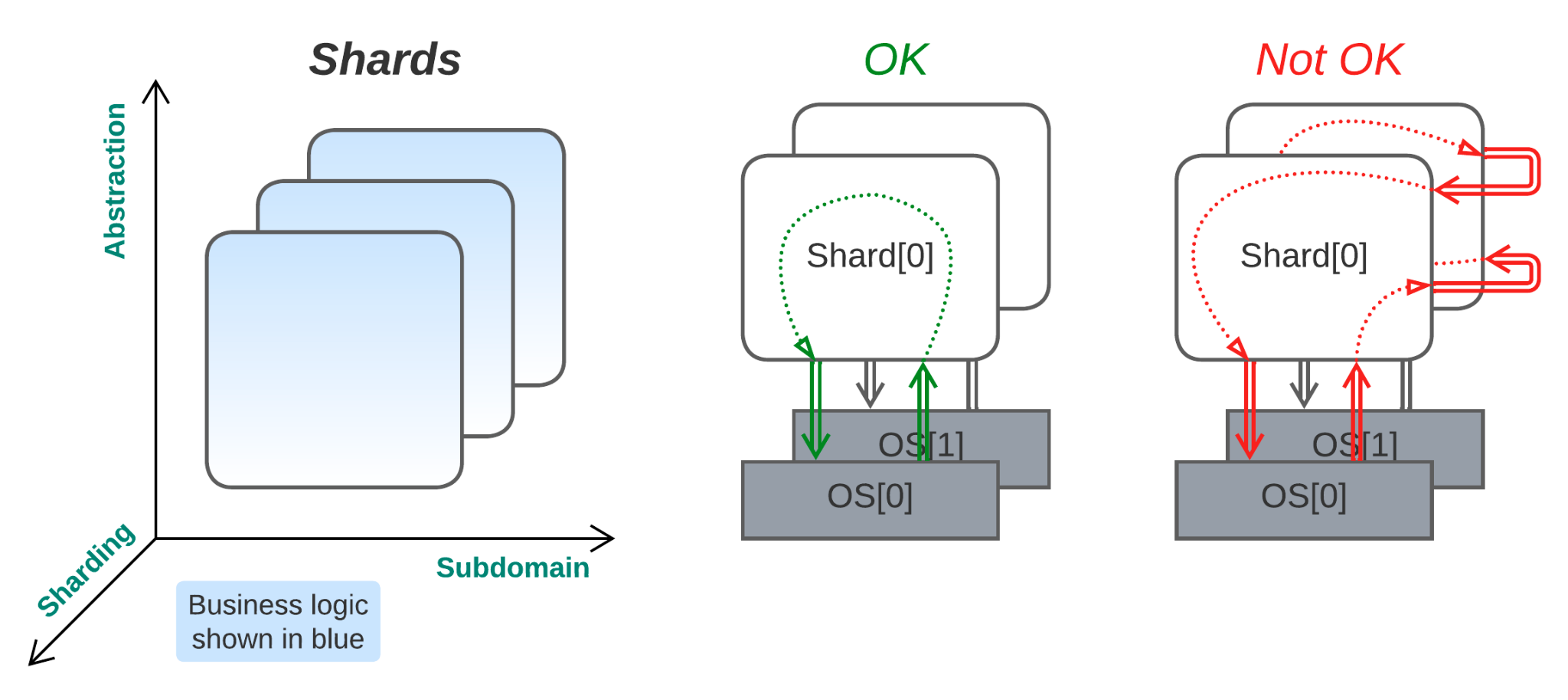
* The vertical axis maps ***abstraction*** – upper parts are more abstract (high-level business logic in Python or DSL), whereas lower parts are implementation-specific (device drivers in C or highly optimized library data structures). Users pay for the higher levels of the software and don’t care about the low-level implementation unless something there goes wrong. The upper modules rely on the lower ones.
* The horizontal axis resolves ***subdomains*** in an arbitrary order. Moreover, the subdomains along the axis may vary (belong to different domains) at different heights (abstraction levels); for example, at the lower (OS) level, there is network, HDD, video, sound, mouse and keyboard drivers/interfaces, while the upper application level may be concerned with players, monsters and spells, and on top of it resides yet another layer with metadata: game scores, achievements, player rooms, etc. These different dimensions of subdomains should have been shown along dedicated orthogonal axes, but the drawing has only two available dimensions, making it necessary to map all the subdomains to a single axis of the diagram. Nevertheless, this does not make the diagrams overly messy, as the different domains are usually found at diverse heights (abstraction levels mapped to layers of the system).
* The Z-axis (which is hard to show in 2D and is thus directed diagonally) corresponds to ***sharding*** and shows multiple instances of modules. It is important for a few of the structures discussed and is omitted in other cases.

I believe such coordinates are [common](https://en.wikipedia.org/wiki/Glibc#/media/File:Linux_API.svg) in [OS](https://en.wikipedia.org/wiki/Rump_kernel#/media/File:OS_rumparch.png) and embedded systems diagrams, but I have never seen the axes being explicitly named.

Sample diagrams of various basic shapes in an *ASS* coordinate system will be analyzed by deducing the system properties from the structural diagrams and matched to well-known architectural patterns, with some coverage of common variants.

In some cases, it will be convenient to weaken the actors model by also covering synchronous interactions (RPC or direct method calls) between loosely coupled modules so as to include more examples.

## Shards / Instances



***Multiple instances***. A monolith is cut into functionally identical pieces that usually don’t interact. Alternatively, several identical instances (*shards*) of a monolith are combined into a system to serve user requests together.

*Why is inter-instance communication avoided? Because of possible race conditions (shown in the figure in red): if there were a task in shard[0] that needed shard[1] to change its data and another task in shard[1] that needed shard[0] to make changes, the system would struggle to stay consistent, in accordance with the CAP theorem [*[*DDIA*](#kix.q1vb5yx6icby)*]. This is similar to deadlocks in synchronous systems; the preconditions for both handicaps are loops in control- or data-flow, caused by flows of opposing directions. As the creation of stable and evolvable systems involves avoiding handicaps, direct instance communication should be avoided.*

*If everything flows in the same direction (Pipeline [*[*POSA1*](#wuckyka5hvdv)*], discussed below), there are no opportunities for concurrent updates that could bring in inconsistency. If all the related data is kept together (Layers [*[*POSA1*](#wuckyka5hvdv)*], the next section), loops are also avoided, and the system state is consistent.*

*The best case is always the one that does not need processing. The green scenario in the figure above is very close to that: the shard gets a request, does some internal magic, and returns the response without having to deal with any outside modules, other shards included. This is possible if the shard owns all the information required for processing the events it gets assigned.*

*Benefits* (on top of those for *Monolith* from Part 2):

* Sharding allows for nearly unlimited (being wary of the size of the dataset and traffic bottlenecks) scalability under load (by spawning more instances).
* Running multiple instances of an (ordinarily stateless) actor allows for synchronous calls (*Reactor* [[POSA2](#kix.hefqusybmb4l)] from Part 2), simplifying the implementation of the business logic. The fact that the actors block for the duration of running tasks is compensated for by the number of actor instances.
* Fault tolerance – even if one instance crashes, the system still survives, and the unserved request may be forwarded to other instances, *except* when the crash is reproducible – in that case, the request will crash all the actors it is forwarded to.
* Support for [*Canary Release*](https://martinfowler.com/bliki/CanaryRelease.html) – it is possible to deploy a few instances of a new version of an actor for testing in parallel to the main workforce of a previous stable version.

Drawbacks (on top of those for *Monolith* from Part 2):

* Sharing data between shards is slower and gives rise to problems. Usually, the shards don’t know about each other, but see *Leader/Followers* below and *Mesh* in Part 5 for cases that rely on inter-shard communication. This drawback does not affect *stateless* actors.
* There must be an entity that dispatches requests to shards; thus, this entity is a single point of failure.
* Deploying and managing multiple instances requires an extra administrative effort.

*Evolution*:

* If there appear new use cases that require a shard to access data that belong to other shards, a *shared repository* (Part 4) is the most obvious solution. The structural changes needed may involve unsharding the entire application back to *Monolith* (Part 2), splitting the database into a shared layer (the proper *Shared Repository* pattern from Part 4), or running the shards on top of a *Space-Based Architecture* framework (Part 5). Yet another option is *Nanoservices* (see below).
* If in need of fine-grained scalability, splitting into *Nanoservices* may be the best option for projects with small codebases, while for larger projects *(Micro-)Services* (see below) or *Service-Oriented Architecture* (described in Part 5) are more suitable.
* Fault tolerance may be achieved by [replacing parts of the monolith](https://hillside.net/plop/2020/papers/yoder.pdf) with *Nanoservices* under *Microkernel* (Part 4).
* Getting out of *monolithic hell* [[MP](#kix.hccjphx9ia0l)] can likely be achieved through sharded *(Micro-)Services*, in a way similar to that for unsharded monoliths.

*Summary*: sharding is a mostly free opportunity to scale a monolith to support higher load for isolated users or datasets of moderate size (before the database management or traffic control becomes non-trivial), though for larger systems, *Mesh* (from Part 5) may be more appropriate. However, sharding is applicable only in cases where incoming requests may be binned to independent groups (i.e., no requests from one group touch the state that any other group writes to).  
*Example*: a network file storage where every user gets their own folder. A given folder’s URL maps to a single shard responsible for serving the user.

The code for shards is as simple to start writing as that for a monolith, and it faces similar threats, namely *monolithic hell* [[MP](#kix.hccjphx9ia0l)], as the project grows.

*Common names*: **Instances**.

*Software architecture*: **Pooling** [[POSA3](#g9gu50wy8qdn)].

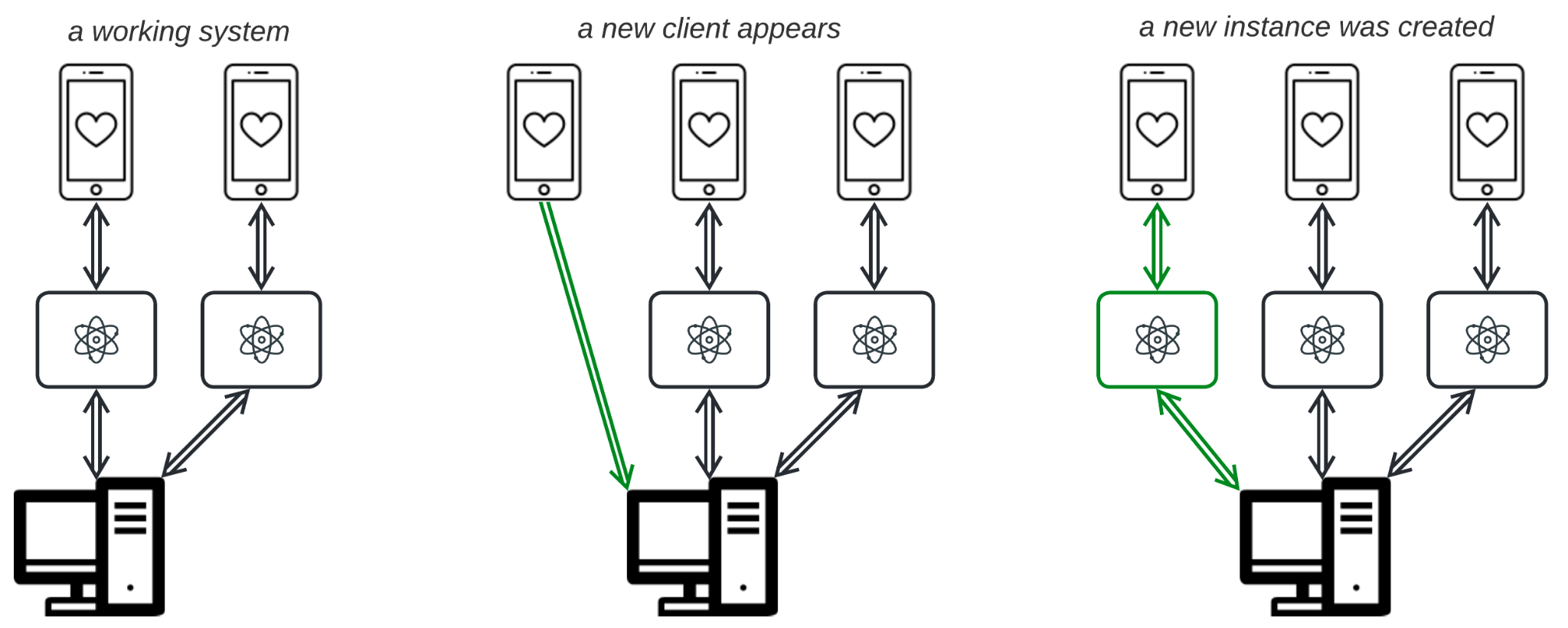
*System architecture*: **Sharding**,CGI/fCGI, FaaS.

Sharding is applicable to both monolithic applications and individual services or actors in more complex asynchronous systems. The latter case helps to remove bottlenecks in the system by creating more instances of the most heavily loaded modules than of modules under lower load, which results in an optimal use of resources.

Actor (or service) instances in a sharded architecture resemble execution threads in a multithreaded reactor. The difference between threads and processes is the source of most of the *Shards’* benefits and drawbacks compared to those of a multithreaded *Reactor* (Part 2).

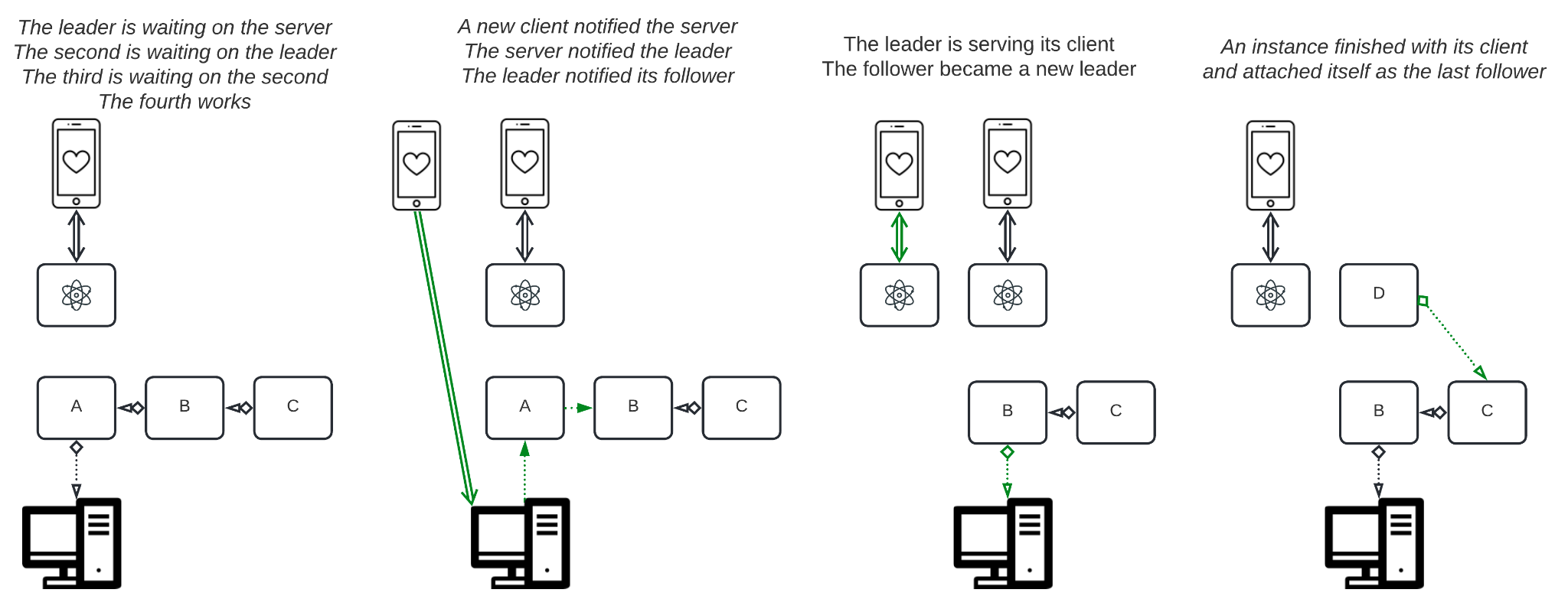
I remember the next three approaches to dispatching the work among threads/instances:

### Create on Demand (elastic instances)



The load (e.g. the number of simultaneous users for instance-per-user model) is unpredictable, or service instances may run on the client side. Create an instance as soon as a new client connects and delete it upon disconnect. This may be quite slow because of the instance creation overhead, and peak load may consume all the system resources, so new users will not be served.  
*Examples*: frontend, call objects in a telephony server, user proxies in multiplayer games and coroutines in *Half-Sync/Half-Async* from Part 2. A stateless multithreaded *Reactor* (Part 2) also fits the definition of *Shards*, as threads cannot change each other’s state – this is yet another case of a system that matches the descriptions of several patterns.

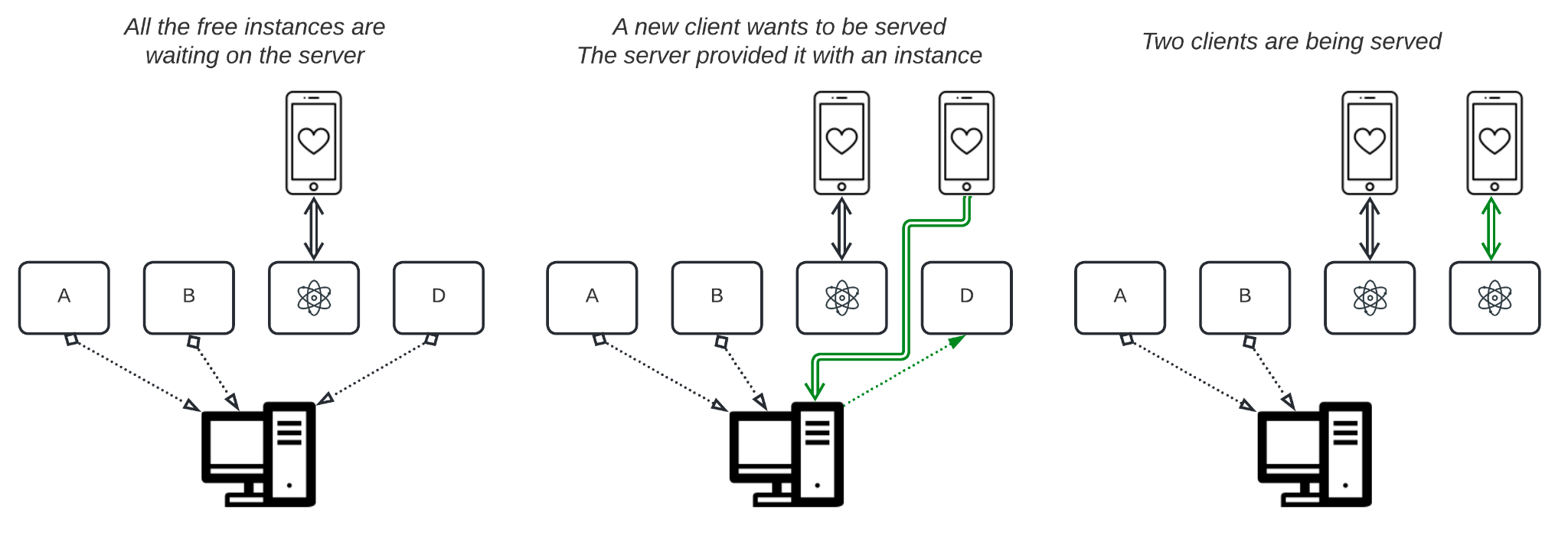
### Leader/Followers [[POSA2](#kix.hefqusybmb4l)] (self-managing instances)



The instances are pre-initialized (*pooling*) and linked into a list. One instance (the *leader*) is waiting on a socket, while other instances (*followers*) are idle. As soon as the leader receives a request, it yields the socket to its first follower and starts processing the popped message. The follower that receives the socket becomes the new leader and starts waiting on the socket for an incoming request. As soon as one of the old leaders that were processing messages finishes its work, it joins the end of the followers queue.

This makes some sense but does not scale between servers. Thus, a hardware failure will kill the entire system.

### **Load Balancer, aka Dispatcher** [[POSA1](#wuckyka5hvdv)] **(**external dispatch**)**



Some external service (Nginx or a hardware device) dispatches queries over an instance pool. This approach is scalable over multiple servers, but the balancer is the single point of failure.

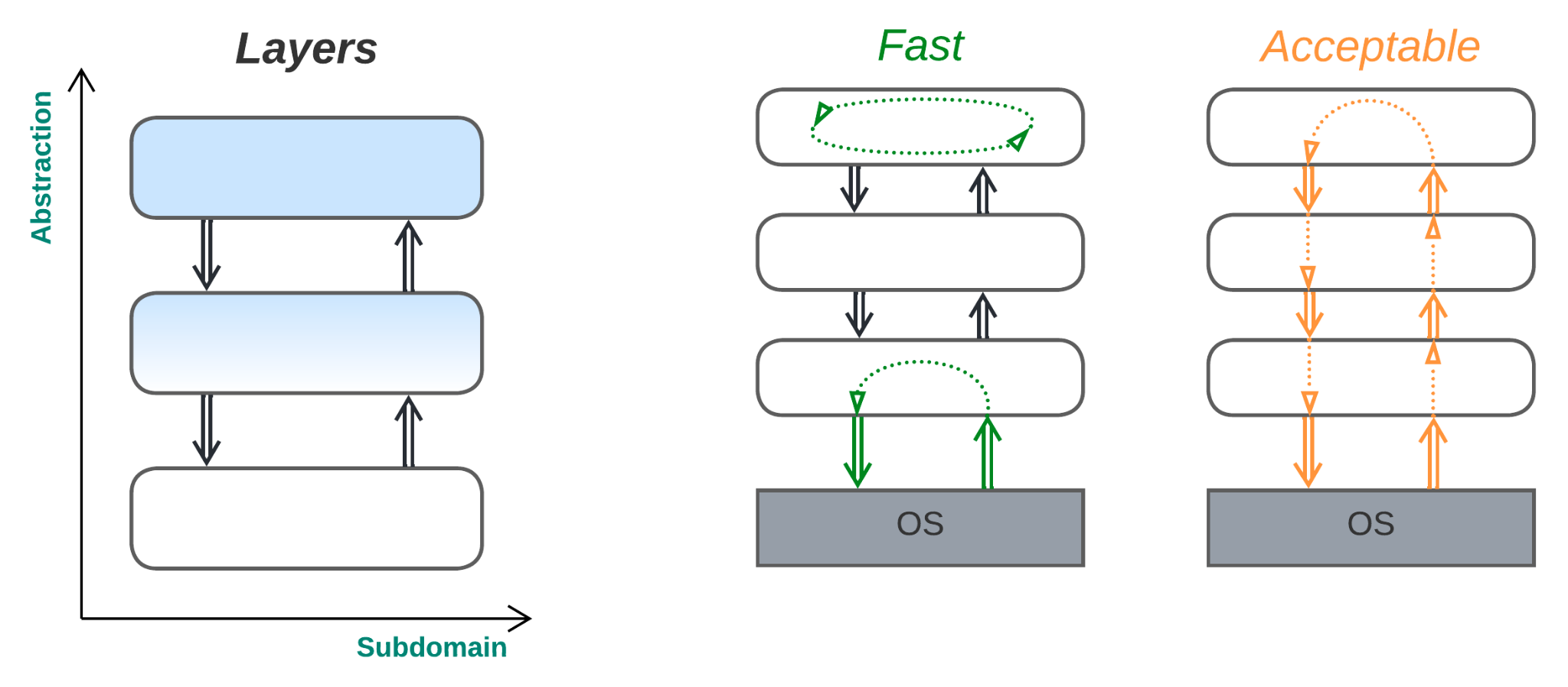
Actually, this is a *layered* system (see below), as the dispatcher and the shards belong to different abstraction levels; the dispatcher is concerned with connectivity (low abstraction: bytes, protocols, hardware) while the shards perform the business logic (high abstraction: users, goods, ads).

### Mixtures

Again, a couple of mixed cases are possible, e.g. when:

1. A *Load Balancer* dispatches requests among servers that run *Leader/Followers* queues.
2. All the instances on a server are waiting on an OS or framework object (e.g. a socket). An incoming request is assigned to one of the waiting instances. In this case, the *Load Balancer* is implemented by the OS or the framework.

## Layers [[POSA1](#wuckyka5hvdv)]



***Separate the business logic from the implementation details****.* The monolith is cut by messaging interfaces into pieces that model different abstraction levels. The uppermost level makes strategic decisions, whereas the lowest layer deals with the hardware.

*Benefits*:

* All the subdomains in any given layer are cohesive (accessible via synchronous calls). For example, in a game, the code for a unit may easily get information about its environment and interact with other units. This also makes debugging markedly easier unless the control-/data-flow under investigation leaves the current layer.
* The high-level logic is decoupled from the low-level code, e.g. a unit’s AI planner does not care about the unit’s rendering, as it is encapsulated with a messaging interface. This should (in theory) provide for more compact and readable code.
* The high-level logic and the low-level code may have different properties (*quality attributes* or *“-ilities”* [[MP](#kix.hccjphx9ia0l)]) to satisfy conflicting *non-functional requirements*. A unit’s AI may be busy planning (which is a single long calculation) its future actions for a couple of seconds, while a renderer should draw (a periodic real-time task) the unit every 15 ms.
* All the layers may be executed in parallel (a layer per CPU core). However, the load distribution is usually quite unbalanced, making whatever benefit that is gained insubstantial.
* Layers that don’t possess a system-wide state may be independently sharded under load. For example, it is hard to shard a map (which is a unique shared resource), but it is practicable to shard the units’ AI planners (as they don’t interact directly).
* The layers may be deployed independently (take OS updates as an example), though not without some [effort](https://kb.netgear.com/21853/What-is-the-dual-image-feature-and-how-does-it-work-with-my-managed-switch) in terms of architecture and implementation.
* Layers support fast and easy processing of events that don’t involve multiple layers (e.g. the renderer may render the last cached state of the world without any need for direct access to the full units or map data).

*Drawbacks*:

* It is quite inconvenient to code and debug use cases with logic that is spread over multiple layers. In practice, the entire business logic often resides in the topmost layer or is divided between a UI/metadata layer that governs the system’s behavior and the domain model layer that implements most of the interactions (see *Application Service*, *Plug-ins*, *DSL* and *Orchestrators* in Part 4 of this series).
* Responsiveness and efficiency deteriorate if an incoming event has to traverse several layers to be handled. In practice, lower layers tend to combine several low-level system events into larger and more meaningful app-level events, decreasing the communication overhead towards the business logic layer(s).
* As the project evolves, one of the layers (usually the uppermost one) tends to grow out of control. The application of the *Layers* pattern does not efficiently reduce the project’s complexity, though it may help with implementing the business logic in relatively high-level terms.

*Evolution*:

* The most common issue with layered structures is *monolithic hell*, when one of the layers grows too large for the developers to understand. Getting out of it requires splitting the overgrown layer(s) into services. The resulting system will usually consist of *(Micro-)Services* (see below), often featuring *Application Service* or *Shared Database* (both described in Part 4). Other options include *Pipeline* (described below), which is applicable to data processing systems, *Service-Oriented Architecture* and *Hexagonal Hierarchy* (both from Part 5).
* Often, the business logic must be protected from external dependencies. This is achievable with *Hexagonal Architecture* (Part 4) or *Pipeline*.
* Customization and metadata are available by applying *Plug-Ins* or *Domain-Specific Language* (both from Part 4).
* Satisfying contradictory non-functional requirements requires splitting the business logic into multiple services. *Shared Repository* (Part 4) is likely to be the first step of the process.
* Scalability and fault tolerance will likely require sharding, with a subsequent transition to *Service-Oriented Architecture* (Part 5) or *Nanoservices* being possible if needed.

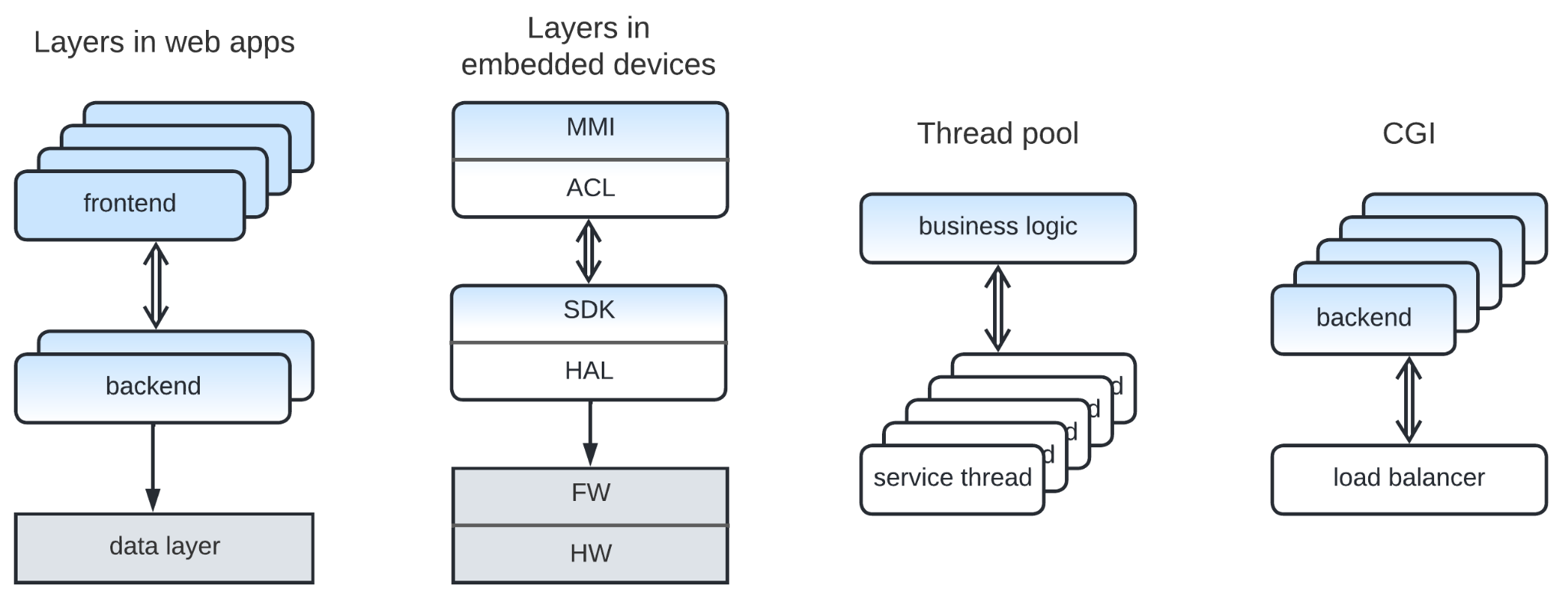
*Summary*: a cheap division of labor for cases with cohesive code at each abstraction level, while the levels themselves are loosely coupled (like game logic and renderer). *Layers* are ubiquitous as a convenient approach which is hard to misuse.

*Common names*: **Layers** [[POSA1](#wuckyka5hvdv)]*.*

*System architecture*: **n-Tier**.

Splitting into layers, like sharding, can be applied to monoliths, to individual actors or services, and even to whole systems containing multiple asynchronous components (e.g. *SOA*, *Mesh* and *hierarchical* architectures from Part 5). In other cases, prominent throughout Part 4, a monolithic layer is added to an existing system, usually to improve communication between its components. On rare occasions, subdomains may vary in the number of layers employed, the most common example of such a case being [CQRS](https://herbertograca.com/2017/10/19/from-cqs-to-cqrs/), where the command path uses a domain model layer (often with DAO) that is totally absent from the query path (which may sometimes consist of a barebone SQL editor).

As it usually happens, there are variants or corner cases:



### Multi-tier System

A system is usually divided into layers/tiers according to the level of abstraction. With *3-tier* web services, users face the highest abstraction *frontend* layer, where domain concepts are represented with text, input fields, buttons and animations without heavy mathematical calculations. The next layer, *backend*, contains inheritance trees, if…elseif...else decisions, algorithms and containers embedded in *domain objects* [[DDD](#kix.lip8q7q2gstj)] and *use cases*. Below that is a *data layer* that contains tables and indices optimized for efficiency at the cost of resemblance to the domain entities. Every layer is very likely to be implemented in a dedicated programming language which better fits the layer’s needs (or which the available programmers are familiar with). The separation of implementations, including the choices of languages, is achieved via language-independent interfaces (e.g. HTTP and SQL for most of 3-tier systems, libc API and syscalls for Linux OS).

Externally-defined interfaces allow for layered architectures to apply equally well to objects, actors and mixed systems; a frontend may communicate with a backend asynchronously (in an actor-like way), while the backend makes synchronous RPCs to the data layer.

A typical 3-tier system may use several sharding options at once: the instances of the frontend layer are dynamically created on end-user devices, while the backend relies on a load balancer and an instance pool, albeit with the data layer left unsharded (a single-box DB instance).

The backend load balancer has been excluded from the figure, as it is a minor part of the system (it does not implement any business logic) and there was no convenient place for it in the drawing. However, it could have been shown between the frontend and the backend instances intercepting their communication channels, or at the very bottom as an entity with no business logic. Some details will be skipped from now on, as *too much information is no information.*

Layered actors may be used in embedded programming to split the behavior of a control system into a long-term strategy and real-time reflexes, which are commonly programmed into separate chips. In telecom client devices, there are MMI (user interaction and high-level use cases), SDK (low-level use case support and common services) and FW (hardware-specific code) components.

### Thread Pool

A common pattern for offloading long calculations from a main business logic module to service threads that:

* May load all the available CPU cores with the calculations.
* May run in lower priority to avoid starving the main business logic thread(s).
* Run the assigned task to completion, then notify the task creator asynchronously.
* Accept any tasks, as the service threads don’t have any state or business logic of their own.

*Thread Pool* looks like *Layers* + *Sharding*. The management layer that runs the business logic creates and dispatches tasks while the service threads act as mere wrappers for the hardware’s CPU cores.

### Application Pool

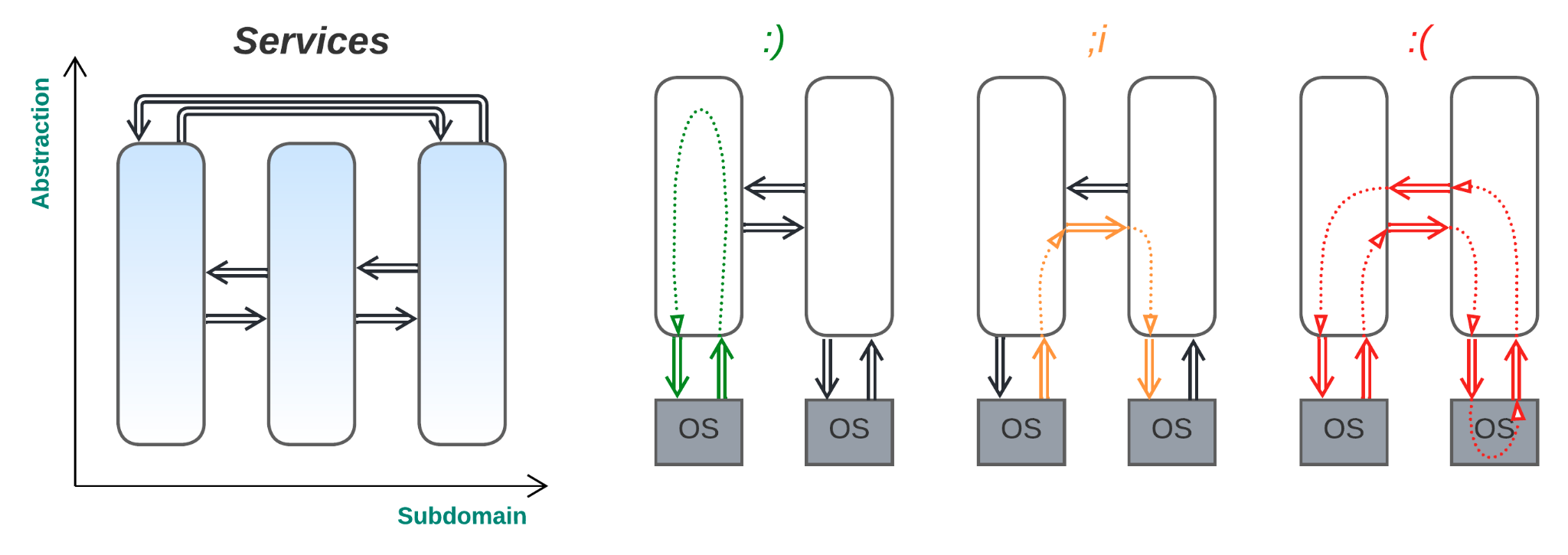
This structure is a mirror reflection of *Thread Pool*. Instead of dumb threads ready to help with any tasks, *Application Pool* manages instances that contain the entire application’s business logic, each serving user requests.

The *coroutines* of *Half-Sync/Half-Async* (from Part 2) may be considered application instances over a shared service layer; even the “Half-/Half-” in the pattern’s name evokes the notion of layers. In fact, the pattern itself originally described the kernel and user space layers of OSes.

A backend with a *load balancer,* described in the *Shards* section above, is also an example of the *Application Pool* architecture; there is a minor low-level component nearly devoid of business logic that governs high-level application components (CGI).

Yet another example may probably be found in *orchestrator* [[MP](#kix.hccjphx9ia0l)] (see Part 4) frameworks.

## Services



***Divide by functionality***. An initial monolith is cut into actors that cover individual subdomains, with the expectation that most use cases don’t cross subdomain borders.

*Benefits*:

* Decoupling subdomains is the only scalable way to reduce the code complexity, as the division results in pieces of more or less equal sizes.
* Coding and debugging for an individual subdomain is easy, even when the high-level logic is strongly coupled to low-level details.
* The subdomains’ logic is kept loosely coupled for the duration of the project (i.e. modularity is not violated), as it is extremely hard to hack around a messaging interface even under pressure from management.
* The subdomains are decoupled by properties: one actor may be implementing real-time cases, while for another one, long-running tasks are common.
* All the subdomain implementations run in parallel.
* The subdomains can be sharded independently if the load varies and the subdomain does not have a global state.
* The subdomain actors may be deployed independently.
* Use cases that don’t cross subdomain borders (i.e., are limited to one subdomain) are very fast.

*Drawbacks*:

* Use cases that involve logic or data from several subdomains are very hard to code and debug.
* If several services depend on a shared dataset, the data often needs to be replicated for each service, and the replicas (*views* [[DDIA](#kix.q1vb5yx6icby)]) should be kept synchronized.
* The system’s responsiveness and throughput deteriorate for system-wide use cases (that involve several subdomains).
* Services are left interdependent by contract (*choreography* [[MP](#kix.hccjphx9ia0l)]) unless there is a shared layer for coordinating inter-service communication and system-wide use cases (see Part 4 for multiple examples).
* Integration / operations / system administration complexity emerges.

*Evolution*:

* Should individual services grow too large and complex, causing *monolithic hell* [[MP](#kix.hccjphx9ia0l)], they should be split into smaller services, likely transferring the system to *Cell-Based Architecture* (Part 5). There are also the more dubious options of *Application Service* (Part 4) and *Service-Oriented Architecture* (Part 5).
* Having too many connections between services can be alleviated by introducing *Middleware* (Part 4) or refactoring to *Cell-Based Architecture* (Part 5).
* Protecting the business logic from the environment is achievable in the following ways: the whole system of services can be encapsulated with *Gateway* (Part 4), while *Hexagonal Architecture* (also from Part 4) can be applied to isolate the business logic of the individual services from third-party components.
* The project can often be developed faster when *Shared Repository* or *Middleware* (both from Part 4) is used.
* If the domain is found to be strongly coupled, the services either need to be integrated with a monolithic layer, i.e. *Application Service* with *Orchestrators* or *Shared Repository* (all described in Part 4), or the entire business logic should be merged into *Hexagonal Architecture* (Part 4) or even *Monolith* (Part 2).
* The mutual dependency of services, caused by their reliance on each other’s interfaces and contracts, is reduced by the application of patterns that manage the communication and global use cases. *Middleware* (Part 4) abstracts the transport, *Shared Repository* (Part 4) provides communication via changes in the global state, while *Gateway* (with *Mediators* or *Orchestrators*) or *Application Service* (all described in Part 4) takes control of global use cases, thus often removing the need for direct interservice communication. *Mesh* and *Hierarchy* patterns from Part 5 may completely remove the need for some of the communication aspects in project domains to which they are applicable.
* Extra flexibility and testability can be found with *Pipeline* (see below), but it only fits simpler kinds of data processing.
* High scalability and fine-tuned fault tolerance are achievable with *Nanoservices* (see below) and distributed *Microkernel* (Part 4).

*Summary*: this kind of division is good for systems that feature mostly independent modules; otherwise, the implementation complexity and request processing time increases.  
*Exception*: for huge projects, the division by subdomain is the only way to keep the total project complexity manageable. Even though many use cases become intrinsically more complex to implement and operational complexity may become untrivial, the splitting of a huge monolith into several asynchronous modules still removes much of the accidental complexity in the code and allows the resulting modules to be developed and deployed relatively independently, leading one out of *monolithic hell* [[MP](#kix.hccjphx9ia0l)].

*Other ways to split a huge monolith don’t reduce the code complexity well enough: sharding does not change the code at all (as every instance includes the entire project’s codebase), while layered systems usually contain 3 to 5 uneven layers, with one of the upper layers still growing too large to be manageable and too cohesive to be split again. At the same time, dividing a project by subdomain boundaries may easily result in a dozen loosely coupled services of nearly uniform size, and it may be possible to recursively split those services into subsubdomains if the need arises. The system as a whole becomes slower and more complicated, but the individual components remain manageable. There is another similar, yet better, option, namely hierarchical decomposition (described in Part 5), but it is not applicable in most domains.*

*Common names*: **Actors** (embedded).

*System architecture*: **Microservices** [[MP](#kix.hccjphx9ia0l)].

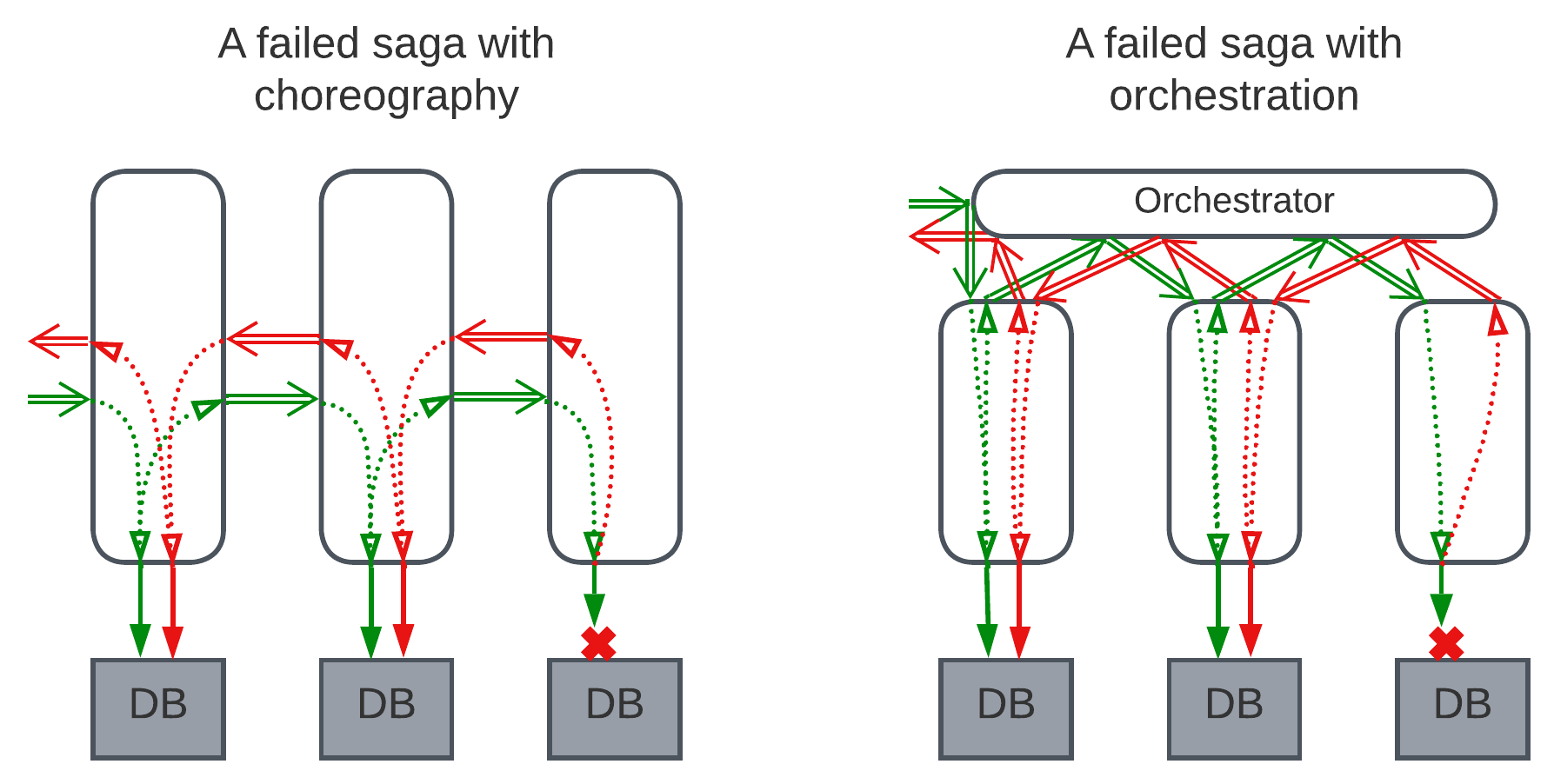
Like the other two basic patterns described in this article, splitting into services applies to both monoliths and individual modules. Three kinds of architectures that result from splitting a monolithic application into services are described in the subsequent sections. Part 4 is dedicated to architectures that retain a monolithic layer together with layers of services, whereas Part 5 deals with fragmented systems where every layer has been divided into subdomain services.

Traditionally, there are variants:

### Microservices (Domain Actors)

***Make subdomains independent****.* If a target domain can be divided into several subdomains, it is likely that the business logic inside each of the subdomains is more cohesive than it is between the subdomains, meaning that any given subdomain deals with its own state and functions much more often than it requires help from other subdomains. Tasks limited to a single subdomain are easy, those chaining (*fire and forget*) or multicasting notifications over several subdomains – acceptable, but once a task requires the coordination of and feedback from several subdomains – everything turns into a mess (see Part 1), as the subdomain actors are too independent. Not only do *views* [[DDIA](#kix.q1vb5yx6icby)] or *sagas* [[MP](#kix.hccjphx9ia0l)] emerge, but the task’s logic is likely to consist of multiple distributed pieces, possibly belonging to several code repositories, which makes troubleshooting the task as a whole very inconvenient, and its execution often inefficient. A system that mostly relies on such coordinated tasks is sometimes called a *distributed monolith* [[MP](#kix.hccjphx9ia0l)] (Part 5) and is usually considered to be the result of an architectural failure, namely incorrect subdomain boundaries or overly fine-grained services [[SAP](#kix.5dxhu7zi5ihm)]. It is recommended to [first build a monolith](https://martinfowler.com/bliki/MonolithFirst.html), then locate loosely coupled modules [[DDD](#kix.lip8q7q2gstj)] (possibly refactoring out redundant dependencies and moving pieces of code between modules), and only after the already running system has converged into a loosely coupled state can it be split into a set of microservices. Another option to reduce a monolith is by [separating the most loosely coupled services](https://hillside.net/plop/2020/papers/yoder.pdf) on an individual basis. This is a lengthy process, but it does not destabilize the whole project.

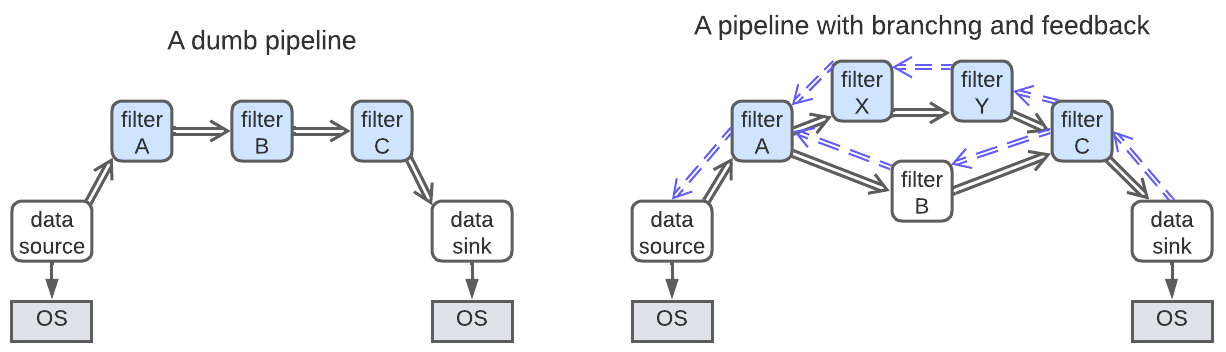
*Domain Actors* / *Microservices* may be used for loosely coupled *data processing* (enterprise) and *control* (telecom) systems; see Part 2 for the distinction.



When things go wrong (as they tend to do in real life), systems start to rely on distributed transactions, causing *sagas* [[MP](#kix.hccjphx9ia0l)] (rules for running system-wide transactions and rolling back failed ones) to be implemented. They may be *choreographed* [[MP](#kix.hccjphx9ia0l)] by hardwiring the task processing steps into services or *orchestrated* [[MP](#kix.hccjphx9ia0l)] by adding an extra business logic layer on top of the services. A choreographed logic is hard to understand, as a single use case is split over multiple code repositories, while an orchestrated logic adds a system-wide layer on top of the services; it is slower because more messaging is involved and may create a single point of failure.

For this and other practical reasons, real-world *microservice* systems are usually augmented with extra layers such as *Gateway* [[MP](#kix.hccjphx9ia0l)], *Orchestrators* [[MP](#kix.hccjphx9ia0l)], *Shared Database*, or *Message* *Broker* [[MP](#kix.hccjphx9ia0l)] – all of which are described in Part 4 of the series, or are *Hierarchical*, with *Hexagonal* or *Cell* subsystems for services (Part 5).

### Pipeline (Pipes and Filters [[POSA1](#wuckyka5hvdv)])



***Separate data processing steps***. A special case of *Domain Actors* (*Broker Topology* [[SAP](#kix.5dxhu7zi5ihm)]), it is very specialized but widely used; while in most other patterns, participants may communicate in any direction, *Pipeline* is limited to unidirectional data flow – every actor (called a *filter* in this pattern) receives its input data from its input event pipe(s), processes the data and sends results to its output message pipe(s). A filter is usually responsible for a single step of the data processing and is kept small and simple.

The filters can be developed, debugged, deployed and improved independently of each other [[DDIA](#kix.q1vb5yx6icby)] thanks to their identical interfaces (though input data type tends to vary) – in fact, the filters are completely ignorant of each other’s existence, a nearly unachievable goal for microservices. Event replay (with a data sample from a pipeline) is used for profiling and for the comparison of outputs between versions of a filter [[DDIA](#kix.q1vb5yx6icby)]. Pipelines sometimes branch and in rare cases may change topology at run time. Pipelines are usually created by a factory during system initialization, often based on a configuration loaded from a file.

This pattern is only useful in *dataflow*-dominated systems (as *control* systems both tend to change behavior at run time and usually need real-time responsiveness – see Part 2). Nevertheless, if *Pipeline* fits the project’s domain requirements, it may be used without much further investigation, as the pattern is very flexible and simple. However, there are some domains dominated by data processing delay (instead of the more common throughput objective), namely night vision, augmented reality and robotics, where pipelines are inappropriate solely because the pattern is not optimal for fast response time.

*Benefits*:

* It is exceedingly easy to develop and support.
* The filters run in parallel, allowing the OS to distribute the load over the available CPU cores.

*Drawbacks*:

* If some of the filters are slower than others, there is a chance of the data overrunning the pipe’s capacity and starving the OS of RAM when under heavy load. This can be alleviated by implementing a feedback engine, which, however, complicates the simple idea of pipelines.
* *Pipes and Filters* are likely to suffer from copying the data more often than strictly necessary, and the pattern is not well suited for the parallelization of data processing inside individual filters. This means that a decently written *Monolithic* application (probably using a *Thread Pool*) will process each data packet faster, and is very likely to have better throughput.

*Evolution*:

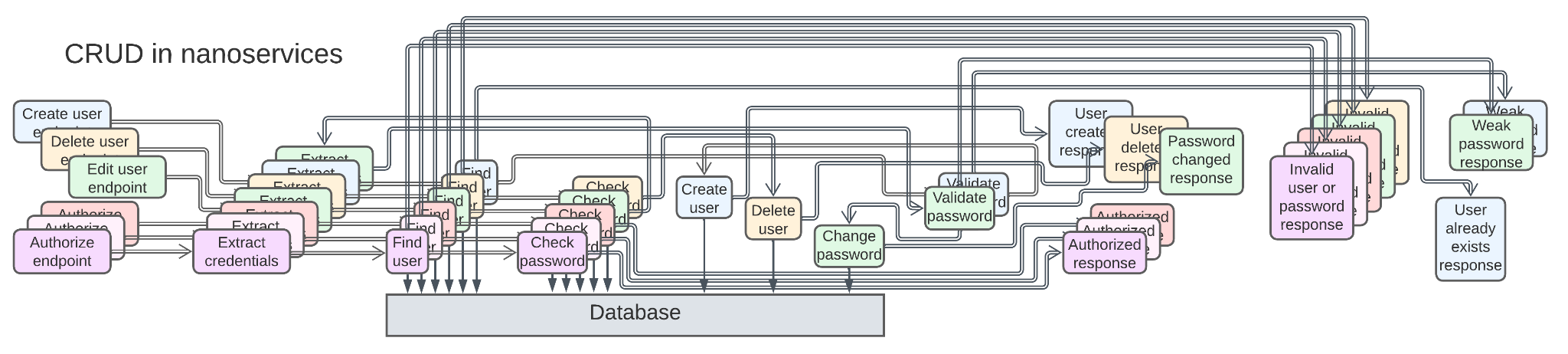
* Fine-grained scalability and fault tolerance may require a transition to *Nanoservices* in a *Microkernel* environment (Part 4).

*Common names*: **Pipes and Filters** [[POSA1](#wuckyka5hvdv)].

*System architecture*: **Pipelines** [[DDIA](#kix.q1vb5yx6icby)].

Implementing feedback ([*congestion control*](https://en.wikipedia.org/wiki/TCP_congestion_control)) for highly loaded pipelines may require the filters to send back a confirmation for every processed packet, the number of packets in a pipe, or the number of packets that have already been processed. This effectively creates a system where *data* and *control* (feedback) packets flow in opposite directions. Another feedback option is to add a supervisor *middleware* (Part 4) that collects throughput statistics and manages the quality/bitrate settings of the filters to avoid overloading the pipeline*.* A third feedback option is to make writing to a pipe block if the pipe is full [[POSA1](#wuckyka5hvdv)].

### Nanoservices (Event-Driven Architecture [[SAP](#kix.5dxhu7zi5ihm)])



***Fine-grained actors for supreme fault tolerance, flexibility and scalability***. *Event-Driven Architecture* is an umbrella term for approaches that use some of the following features:

* An actor (*event processor*) for each data processing step (stateless) or domain entity (stateful): the proper *Nanoservices*
* *Publish-Subscribe* communication
* Pipelined (unidirectional) and often forked flow that tends to involve persistence (database access, event sourcing or a built-in framework capability)

Strictly speaking, *Microservices* and *Pipelines* are also *Event-Driven Architectures*, but the name is often used for *pipelined pub/sub nanoservice systems*.

Like in *Pipeline*, an external event passes through multiple steps that transform it, usually by adding more data. In contrast to *Pipeline*’s *filters*, *event processors* may mark the event as disallowed to break the normal processing chain, and they tend to read from or write to **shared** databases. Unlike with *Pipelines*, multicasts, either via pub/sub subscriptions or by *Mediator* [[SAP](#kix.5dxhu7zi5ihm)] (Part 4), are common.

*Choreography* [[MP](#kix.hccjphx9ia0l)] (*Broker Topology* [[SAP](#kix.5dxhu7zi5ihm)]) and *orchestration* [[MP](#kix.hccjphx9ia0l)] (*Mediator Topology* [[SAP](#kix.5dxhu7zi5ihm)]) apply, just like for *microservices*. Unlike *microservices*, *event processors* are very fine-grained, usually containing code for a single action or domain entity, and are spawned in large numbers.

While most of the system architectures feature four levels to distribute the domain complexity over, namely: lines of code in methods, methods in classes, classes in services, and services in the system (see Part 1 for the relevant discussion), *nanoservices* feature only the first and the last means, as a nanoservice implements a single method. Therefore, this pattern is not applicable for complex domains.

*Benefits*:

* Very good fault tolerance is achieved, as every part of the business logic and every user request is isolated.
* *Nanoservices* run in parallel, allowing the OS to distribute the load over CPU cores or a load dispatcher to spread requests to multiple servers.
* *Nanoservices* are nearly perfectly scalable unless they access a shared database (which they do) or saturate the network bandwidth / run out of money in cloud deployments.
* Individual *nanoservices* are easy to develop and support.
* Individual *nanoservice* implementations may be reused.
* The system is evolvable by inserting new *nanoservices* or rewiring the existing ones unless it grows too complex to grasp all the interactions and contracts.

*Drawbacks*:

* Integration complexity may quickly overwhelm the project.
* Debugging and testing will not be easy.
* Any coordinated database writes will either limit the system to a single shared database or require distributed transactions or *sagas* [[MP](#kix.hccjphx9ia0l)], increasing complexity and limiting scalability.
* Component reuse is not always going to work because contracts may vary.
* Too much communication wastes system resources, often increases response times and limits scalability.

*Evolution*:

* *Microkernel* frameworks (from Part 4) allow for a fast start on the project and the handling of failure recovery in the application code.
* *Space-Based Architecture* (from Part 5) may help to distribute the shared repository.

*Common names*: Actors (telecom), Flowchart.

*System architecture*: **Nanoservices**, Event-Driven Architecture [[SAP](#kix.5dxhu7zi5ihm)].

*Nanoservices* provide good flexibility and scalability for simple systems with few use cases but tend to turn into an integration nightmare as the functionality grows. Moreover, the almost inevitable use of [shared databases](https://peterdaugaardrasmussen.com/2019/08/14/what-is-a-nano-service/) (as one nanoservice is dedicated to creating users, another to editing them, and the third and fourth to deleting and authorizing them, respectively) limits most of the benefits. The system diagram looks very similar to a *flowchart*; ultimately, that’s likely what stateless nanoservices are, except that flowcharts for all the implemented use cases coexist in the production system. This assumption is confirmed by the visual programming tools (like jBPM) being advocated [[SAP](#kix.5dxhu7zi5ihm)] for nanoservices.

Distributed actors frameworks, e.g. Akka or Erlang, are often used for stateful *Nanoservices*, providing a fault-tolerant *Microkernel* environment (described in Part 4).

## ASS Compared to Scale Cube

[[MP](#kix.hccjphx9ia0l)] defines *scale cube* as a means of scaling an application in the following ways:

1. Cloning instances (which corresponds to sharding a backend in ASS)
2. Functional decomposition (splitting a monolith into microservices)
3. Data partitioning (which corresponds to sharding a data layer in ASS)

The ASS diagrams used throughout the current series of articles represent *evolvability* (how easy it is to change the system) through decoupling:

1. Decoupling by **A**bstraction level
2. Decoupling by **S**ubdomain
3. **S**harding

Two dimensions of ASS (abstraction and subdomain) correspond to the functional decomposition of scale cube, while both the cloning and the data partitioning of scale cube match to ASS’s sharding at different abstraction levels.

[[DDIA](#kix.q1vb5yx6icby)] mentions *scaling up* (upgrading CPU, RAM and disk space) vs *scaling out* (going distributed).

Any conclusions? It seems that the coordinate system used in this cycle is unrelated to the scaling coordinates, except for the fact that every diagram is limited to 2 or 3 dimensions. Nevertheless, some kind of sharding is present in all the coordinate systems reviewed.

## Summary:

All the possible ways of splitting a monolith along one of the ASS dimensions were described, creating:

* *Shards* from spawning monolithic instances.
* *Layers* from slicing by code abstractness.
* *Services* when divided by subdomain.
* *Pipeline* and *Nanoservices* as common specializations of *Services*.

The next parts will cover divisions along both abstraction and subdomain dimensions.

## References

* 1. [DDD] Domain-Driven Design: Tackling Complexity in the Heart of Software. *Eric Evans. Addison-Wesley (2003).*
  2. [DDIA] Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann. O’Reilly Media, Inc. (2017).*
  3. [EIP] Enterprise Integration Patterns. *Gregor Hohpe and Bobby Woolf. Addison-Wesley (2003).*
  4. [MP] Microservices Patterns: With Examples in Java. *Chris Richardson.* *Manning Publications (2018)*.
  5. [POSA1] Pattern-Oriented Software Architecture Volume 1: A System of Patterns. *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. John Wiley & Sons, Inc. (1996).*
  6. [POSA2] Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. *Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. John Wiley & Sons, Inc. (2000).*

* 1. [POSA3] Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. *Michael Kircher, Prashant Jain. John Wiley & Sons, Inc. (2004).*

* 1. [SAP] Software Architecture Patterns. *Mark Richards. O’Reilly Media, Inc. (2015).*
  2. *Editor:* [Josh Kaplan](mailto:joshkaplan66@gmail.com)
  3. *Part* 1 2 **3** 4 5
  4. Creative Commons License
  5. This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).